Data Structures with Python (DI03016031)

Complete Lecture Deck – All 45 Lectures

Milav Dabgar

July 25, 2026

Table of Contents I

Data Structure Basic Concepts and Types

Unit 1: Basic Concepts of Data Structures
Lecture 1
Course: DI03016031

Agenda

Introduction to Data
What is a Data Structure?
Need for Data Structures
Classification of Data Structures
Primitive vs Non-Primitive
Linear vs Non-Linear
Static vs Dynamic
Operations on Data Structures
Memory Layout
Examples in Python
Summary

Introduction to Data and Information

Data: Raw, unorganized facts that need to be processed.
Information: Processed, organized data presented in a meaningful context.
In computer science, data is represented as binary numbers.
Data types define the nature of data and operations that can be performed on them.

What is a Data Structure?

A **Data Structure** is a specialized format for organizing, processing, retrieving and storing data.
It provides a means to manage large amounts of data efficiently.
Algorithms + Data Structures = Programs.
It defines the logical or mathematical model of a particular organization of data.

Need for Data Structures

Processor Speed: High-speed processing requires efficient data delivery.
Data Search: Searching in massive datasets requires structured organization.
Multiple Requests: Handling simultaneous user requests efficiently.
Optimizing storage space and retrieval time.

Classification of Data Structures

Data structures can be broadly classified into two categories:
1. **Primitive Data Structures**: Built-in data types provided by the language.
2. **Non-Primitive Data Structures**: Derived from primitive data types. Further classified into Linear and Non-Linear, Static and Dynamic.

Primitive vs Non-Primitive

Primitive:
- Basic building blocks.
- Examples: ‘int‘, ‘float‘, ‘char‘, ‘boolean‘.
- Directly operated upon by machine-level instructions.
Non-Primitive:
- Complex structures.
- Emphasize structuring of a group of homogeneous or heterogeneous data items.
- Examples: Arrays, Lists, Files.

Linear vs Non-Linear Data Structures

Linear Data Structures:
- Elements are arranged in a sequential manner.
- Every element has a unique predecessor and successor.
- Examples: Arrays, Linked Lists, Stacks, Queues.

Non-Linear Data Structures:
- Elements are arranged in a hierarchical or interconnected manner.
- An element can have multiple predecessors or successors.
- Examples: Trees, Graphs.

Static vs Dynamic Data Structures

Static Data Structures:
- Size is fixed at compile time.
- Memory allocation cannot be changed during runtime.
- Example: Fixed-size arrays.

Dynamic Data Structures:
- Size can shrink or expand at runtime.
- Memory is allocated dynamically (e.g., using pointers/references).
- Example: Linked Lists.

Basic Operations on Data Structures

1. **Traversing**: Accessing each element exactly once.
2. **Searching**: Finding the location of a given element.
3. **Inserting**: Adding a new element to the structure.
4. **Deleting**: Removing an existing element.
5. **Sorting**: Arranging elements in a specific order.
6. **Merging**: Combining two data structures into one.

Data Structures in Python

Python abstracts many complex data structures into easy-to-use built-ins:

```
# Primitive-like (Immutable)
a = 10
b = 3.14
# Non-Primitive (Linear, Dynamic)
my_list = [1, 2, 3, 4]  # List acts as a dynamic array
my_list.append(5)       # Insertion
# Non-Linear
my_dict = {'a': 1, 'b': 2}  # Hash map / Dictionary
```

Summary

Data structures are essential for organizing data efficiently.
They are broadly classified into primitive and non-primitive.
Non-primitive structures can be linear (arrays, lists) or non-linear (trees, graphs).
Understanding these concepts is crucial for designing efficient algorithms.

Analysis Terms: Time and Space Complexity

Unit 1: Basic Concepts of Data Structures
Lecture 2
Course: DI03016031

Agenda

Why Analyze Algorithms?
What is Algorithm Complexity?
Time Complexity Definition
Space Complexity Definition
Factors Affecting Complexity
Constant vs Variable Time
Constant vs Variable Space
Introduction to Asymptotic Notations
Need for Asymptotic Notations
Comparing Algorithms
Examples
Summary

Why Analyze Algorithms?

Multiple algorithms can solve the same problem.
We need a way to compare their efficiency.
Analysis helps in predicting the resources an algorithm requires.
Primary resources: **CPU time** and **Memory space**.

What is Algorithm Complexity?

Algorithm Complexity is a measure of how the resource requirements of an algorithm scale with the input size.
Input size is usually denoted by n.
We are interested in the rate of growth of the time or space required as $n \to \infty$.
Absolute execution time is hardware-dependent; complexity is a mathematical model.

Time Complexity Definition

Time Complexity is the amount of computer time it takes to run an algorithm to completion.
It is expressed as a function of the input size n: $T(n)$.
It estimates the number of elementary operations performed.
Examples of elementary operations: assignments, arithmetic operations, comparisons.

Space Complexity Definition

Space Complexity is the amount of memory space required by the algorithm during its execution.
Expressed as $S(n)$.
Total Space = Fixed Part + Variable Part.
Fixed Part: Space for code, constants, simple variables.
Variable Part: Space for dynamic allocation, recursion stack space, dynamically sized structures.

Factors Affecting Complexity

- Hardware (processor speed, memory size, disk I/O).
- Operating System overhead.
- Compiler optimization.
- Programming language used.
- **Input Size** (The most significant theoretical factor).

Constant vs Variable Space/Time

Constant: Does not depend on the input size n.
- $O(1)$ time: Accessing an array element by index.
- $O(1)$ space: Using a few loop counters.

Variable: Depends on n.
- $O(n)$ time: Iterating through a list of size n.
- $O(n)$ space: Creating a new list of size n.

Example: Time and Space Complexity

```
def sum_array(arr):
    total = 0        # O(1) space, O(1) time
    for num in arr: # Loop runs n times
        total += num  # O(1) time per iteration
    return total
```

- **Time Complexity**: $O(n)$ because the loop runs n times.
- **Space Complexity**: $O(1)$ auxiliary space because we only use the ‘total‘ variable, regardless of ‘arr‘ size.

Introduction to Asymptotic Notations

Mathematical tools used to describe the asymptotic behavior of functions.
They describe the limiting behavior as the argument tends towards a particular value or infinity.
Help in abstracting away constants and lower-order terms.
Focuses on the **Dominant Term**.

Need for Asymptotic Notations

If an algorithm takes $T(n) = 3n^2 + 5n + 10$ operations:
For large n, $3n^2$ dominates.
The constants (3, 5, 10) vary by machine and compiler.
Asymptotic notation simplifies this to $O(n^2)$.
It allows theoretical comparison of algorithms independently of hardware.

Summary

Time complexity measures execution time as a function of input size.
Space complexity measures memory usage.
Both are essential for analyzing algorithm efficiency.
Asymptotic notations provide a mathematical framework for this analysis.

Big 'O' Notation and Case Analysis

Unit 1: Basic Concepts of Data Structures
Lecture 3
Course: DI03016031

Agenda

What is Big 'O' Notation?
Formal Definition of Big 'O'
Common Time Complexities
Rules of Big 'O'
Why Case Analysis?
Best Case Time Complexity
Worst Case Time Complexity
Average Case Time Complexity
Linear Search Example
Comparing the Cases
Common Pitfalls
Summary

What is Big 'O' Notation?

Big 'O' (Omicron) notation is a mathematical notation that describes the **upper bound** of a function.
It represents the worst-case scenario or the maximum time an algorithm can take.
Used to classify algorithms according to how their run time or space requirements grow as the input size grows.
It ignores constants and lower-order terms.

Formal Definition of Big 'O'

Let $f(n)$ and $g(n)$ be functions mapping non-negative integers to real numbers.
We say $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that:
$0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
Here, $g(n)$ is an asymptotic upper bound for $f(n)$.

Common Time Complexities

- $O(1)$: Constant Time
- $O(\log n)$: Logarithmic Time
- $O(n)$: Linear Time
- $O(n \log n)$: Linearithmic Time
- $O(n^2)$: Quadratic Time
- $O(2^n)$: Exponential Time
- $O(n!)$: Factorial Time

Rules of Big 'O'

1. **Drop Constants**: $O(2n) \rightarrow O(n)$
2. **Drop Non-Dominant Terms**: $O(n^2 + n) \rightarrow O(n^2)$
3. **Multiplication**: Loop inside a loop $\rightarrow$ multiply complexities (e.g., $O(n) \cdot O(n) = O(n^2)$)
4. **Addition**: Sequential loops $\rightarrow$ add complexities (e.g., $O(n) + O(n) = O(n)$)

Why Case Analysis?

The execution time of an algorithm often depends not just on the size of the input, but on the **nature** of the input.
For example, searching for an item is faster if it's the first item rather than the last.
Therefore, we analyze algorithms in three cases: Best, Worst, and Average.

Best Case Time Complexity

The scenario where the algorithm performs the **minimum** number of operations.
Often denoted by Big Omega (Ω).
Occurs when the input data is organized in the most optimal way for the algorithm.
Generally least useful for practical analysis because we cannot guarantee optimal inputs.

Worst Case Time Complexity

The scenario where the algorithm performs the **maximum** number of operations.
Often described using Big 'O' (O).
Occurs when the input data is in the least optimal state.
Most crucial metric: provides a guarantee that the algorithm will never take longer than this.

Average Case Time Complexity

The expected number of operations across all possible inputs.
Often denoted by Big Theta (Θ) if it matches the worst case.
Requires probabilistic analysis: sum of (Probability of input $\times$ Time for input).
Provides a realistic expectation of performance in day-to-day operations.

Example: Linear Search

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

- **Best Case**: Target is at the first index. $O(1)$
- **Worst Case**: Target is at the last index or not present. $O(n)$
- **Average Case**: Target is in the middle. $O(n/2) \rightarrow O(n)$

Summary

Big 'O' notation provides an asymptotic upper bound for algorithm complexity.
Constants and lower-order terms are ignored.
Algorithms are analyzed in Best, Average, and Worst cases depending on input data.
Worst-case analysis is the most critical for system design.

Python Specific Data Structures & Arrays

Unit 1: Basic Concepts of Data Structures
Lecture 4
Course: DI03016031

Agenda

Overview of Python Data Structures
Lists: Dynamic Arrays
List Comprehensions & Memory
Tuples: Immutable Sequences
Sets: Unique Elements
Dictionaries: Key-Value Pairs
Dictionary Implementation details
Choosing the Right Structure
The ‘array‘ Module in Python
Arrays vs Lists
Array Implementation
Summary

Overview of Python Data Structures

Python provides powerful, flexible built-in data structures.
They are highly optimized (implemented in C under the hood).
Main built-ins:
1. **List**: Ordered, mutable sequence.
2. **Tuple**: Ordered, immutable sequence.
3. **Set**: Unordered collection of unique items.
4. **Dictionary**: Unordered collection of key-value pairs.

Lists: Dynamic Arrays

Lists are dynamic arrays; they can grow and shrink in size.
Can hold heterogeneous data types.

```
my_list = [1, 'hello', 3.14, True]
my_list.append(10)   # O(1) amortized
my_list.insert(1, 'world') # O(n)
```

Under the hood, Lists over-allocate memory to achieve $O(1)$ amortized append operations.

List Comprehensions & Memory

List comprehensions provide a concise way to create lists.

```
squares = [x**2 for x in range(10) if x % 2 == 0]
```

Memory Aspect:
- Lists store references to objects, not the objects themselves.
- Deep copy vs Shallow copy is critical when lists contain mutable objects (e.g., nested lists).
- Use ‘copy.deepcopy()‘ for independent nested structures.

Tuples: Immutable Sequences

Tuples are similar to lists but are **immutable** (cannot be changed after creation).
Defined using parentheses '()'.

```
my_tuple = (1, 2, 3)
# my_tuple[0] = 5  # This raises TypeError
```

Why use Tuples?
- Faster iteration than lists.
- Safe from accidental modification.
- Can be used as Dictionary keys (because they are hashable).

Sets: Unique Elements

Sets are unordered collections of unique elements.
Implemented using Hash Tables.

```
my_set = {1, 2, 2, 3}
print(my_set) # Output: {1, 2, 3}
my_set.add(4) # O(1) average time
```

Operations: Fast membership testing ('in' operator is $O(1)$ average), union, intersection, difference.

Dictionaries: Key-Value Pairs

Unordered collection of key-value pairs (ordered as of Python 3.7+).
Keys must be immutable and hashable; values can be anything.

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict['name'])  # O(1) lookup
my_dict['city'] = 'New York'
```

Highly optimized for data retrieval.

Choosing the Right Structure

- Need order and mutability? $\rightarrow$ **List**
- Need order, no changes, or dictionary keys? $\rightarrow$ **Tuple**
- Need uniqueness and fast lookup? $\rightarrow$ **Set**
- Need to map keys to values? $\rightarrow$ **Dictionary**

Choosing the right built-in structure dramatically affects algorithm efficiency.

The ‘array‘ Module in Python

Python lists are arrays of references. This is memory inefficient for large numerical data.
The ‘array‘ module provides space-efficient storage of basic C-style data types.

```
import array
```

Requires a **type code** to specify the data type (e.g., 'i' for signed integer, 'f' for float).

Arrays vs Lists Example

```
import array
# List: stores references, heterogeneous
my_list = [1, 2, 3, 4]
# Array: stores actual values, homogeneous
my_arr = array.array('i', [1, 2, 3, 4])
my_arr.append(5)
```

- **Advantage of Array**: Compact memory footprint.
- **Disadvantage**: Cannot hold different data types, fewer built-in methods compared to lists.

Summary

Python provides versatile built-in structures: Lists, Tuples, Sets, and Dictionaries.
Lists are dynamic arrays of references.
Sets and Dictionaries use hash tables for $O(1)$ average time complexity.
The ‘array‘ module provides true C-style homogeneous arrays for memory efficiency.

Unit 2: Basics of Object-Oriented Programming

Lecture 5: OOP Concepts
Course: Python Programming (DI03016031)

Agenda

1. Procedural vs OOP
2. What is an Object?
3. Four Pillars of OOP
4. Advantages of OOP

Procedural vs Object-Oriented

Procedural Programming:
- Focuses on verbs (functions).
- Top-down approach.
- Data and functions are separate.

Object-Oriented Programming:
- Focuses on nouns (objects).
- Bottom-up approach.
- Data and behavior are bundled together.

What is an Object?

An object represents a real-world entity.
- **State:** Data or attributes (e.g., color, size).
- **Behavior:** Methods or functions (e.g., run, stop).

Objects are instances of classes.

```
# Conceptually
car = {
  'color': 'red', # state
  'drive': lambda: print('Driving') # behavior
}
```

Core Pillars of OOP

1. **Encapsulation:** Bundling data and methods.
2. **Abstraction:** Hiding complex realities.
3. **Inheritance:** Reusing code by extending existing entities.
4. **Polymorphism:** One interface, multiple forms.

Encapsulation - Concept

Encapsulation acts as a protective shield.
Prevents data from being accessed directly by code outside the shield.
Controlled access via getters/setters.
- Promotes data security and integrity.

Abstraction - Concept

Focusing on essential features rather than the background details.
Example: Driving a car.
- You know how to use the steering wheel and pedals.
- You do not need to understand internal combustion or fuel injection.
Reduces complexity and isolates impact of changes.

Inheritance - Concept

Deriving new entities from existing ones.
Establishes an "IS-A" relationship.
- A Dog IS-A Animal.
Promotes code reusability and hierarchical classification.

Polymorphism - Concept

Ability to take many forms.
A single action can behave differently depending on the object.
Example: The ‘+‘ operator in Python.
- ‘1 + 2‘ -¿ ‘3‘ (Integer addition)
- ‘’A’ + ’B’‘ -¿ ‘’AB’‘ (String concatenation)

Advantages of OOP

- **Modularity:** Troubleshooting is easier.
- **Reusability:** Code can be reused via inheritance.
- **Flexibility:** Polymorphism allows flexibility in defining behavior.
- **Security:** Encapsulation hides sensitive data.

Summary

- OOP is a paradigm based on objects.
- Objects bundle state and behavior.
- The four pillars are Encapsulation, Abstraction, Inheritance, and Polymorphism.
- OOP is ideal for complex, scalable applications.

Next Lecture

Topic: 2.2 Class and Object
- Defining classes in Python.
- Creating objects.
- The ‘self‘ parameter.

Unit 2: Basics of Object-Oriented Programming

Lecture 6: Class and Object
Course: Python Programming (DI03016031)

Agenda

1. What is a Class?
2. Defining a Class in Python
3. Creating Objects (Instantiation)
4. The ‘self‘ Parameter
5. Class vs Object Attributes

What is a Class?

A class is a blueprint or template for creating objects.
It defines:
- **Attributes:** The data the objects will hold.
- **Methods:** The operations that can be performed.
A class is an abstract concept; objects are concrete manifestations.

Defining a Class in Python

Use the ‘class‘ keyword.
Class names conventionally use ‘PascalCase‘.

```
class Dog:
    # Class attribute
    species = 'Canis familiaris'

    # Method
    def bark(self):
        print('Woof!')
```

Creating Objects (Instantiation)

To create an object, call the class name like a function.

```
# Instantiation
my_dog = Dog()
your_dog = Dog()

# Accessing attributes and methods
print(my_dog.species) # Outputs: Canis familiaris
my_dog.bark()         # Outputs: Woof!
```

The ‘self‘ Parameter

‘self‘ represents the instance of the class.
It binds the attributes with the given arguments.
Must be the first parameter of any instance method in the class.
Python automatically passes the instance as the first argument when a method is called.

The ‘self‘ Parameter - Example

```
class Person:
    def set_name(self, name):
        self.name = name # Sets instance attribute

    def greet(self):
        print(f'Hello, I am {self.name}')

p = Person()
p.set_name('Alice') # Equivalent to Person.set_name(p, 'Alice')
p.greet()
```

Object Identity

Every object has a unique identity (memory address).
Use the ‘id()‘ function or the ‘is‘ operator.

```
a = Dog()
b = Dog()
print(a is b) # False
print(id(a) == id(b)) # False
```

Even if their attributes are identical, they are distinct objects.

Class Attributes vs Instance Attributes

Class Attributes:
- Shared by all instances.
- Defined outside any method.
Instance Attributes:
- Specific to each instance.
- Defined inside methods, typically ‘__init__‘ (using ‘self‘).

Class vs Instance Attributes - Code

```
class Employee:
    company = 'Tech Corp' # Class attribute

    def set_details(self, name):
        self.name = name # Instance attribute

emp1 = Employee()
emp1.set_details('John')
emp2 = Employee()
emp2.set_details('Jane')

print(emp1.company) # Tech Corp
Employee.company = 'New Tech' # Changes for all!
```

Summary

- Classes are blueprints; Objects are instances.
- Define classes using the ‘class‘ keyword.
- ‘self‘ is essential for instance methods to access object state.
- Understand the difference between class and instance attributes.

Next Lecture

Topic: 2.3 Constructors
- The ‘__init__‘ method.
- Parameterized constructors.
- Object initialization.

Unit 2: Basics of Object-Oriented Programming

Lecture 7: Constructors
Course: Python Programming (DI03016031)

Agenda

1. What is a Constructor?
2. The ‘__init__‘ Method in Python
3. Default/Non-Parameterized Constructors
4. Parameterized Constructors
5. The ‘__new__‘ vs ‘__init__‘ methods

What is a Constructor?

A special method used for initializing objects.
It is invoked automatically when an object of a class is created.
Primary purpose: Assign values to instance variables.
Prevents the need for a separate ‘set_details‘ method.

The ‘__init__‘ Method

In Python, the constructor is named ‘__init__‘.
It belongs to the category of 'magic' or 'dunder' (double underscore) methods.

```
class MyClass:
    def __init__(self):
        # initialization code
```

It does not return any value (implicitly returns ‘None‘).

Default Constructor

A constructor with no arguments (other than ‘self‘).

```
class DatabaseConnection:
    def __init__(self):
        self.host = 'localhost'
        self.port = 5432
        print('Connection initialized with defaults.')

db = DatabaseConnection()
# Output: Connection initialized with defaults.
```

Parameterized Constructor

A constructor that takes arguments to set instance variables dynamically.

```
class Student:
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no

s1 = Student('Alice', 101)
s2 = Student('Bob', 102)
```

Allows creating diverse objects immediately.

Default Argument Values in '__init__'

You can combine default and parameterized constructors using default arguments.

```
class Rectangle:
    def __init__(self, length=1, width=1):
        self.length = length
        self.width = width

r1 = Rectangle()       # 1 x 1
r2 = Rectangle(5)      # 5 x 1
r3 = Rectangle(5, 10) # 5 x 10
```

Can We Have Multiple '__init__' Methods?

Unlike Java or C++, Python does NOT support constructor overloading natively. If you define multiple '__init__' methods, the last one overwrites the previous ones.

```
class Demo:
    def __init__(self): pass
    def __init__(self, val): pass

d = Demo() # ERROR! Missing 'val'
```

Solution: Use default arguments or class methods (factory methods).

The ‘__new__‘ Method (Advanced)

‘__new__‘ is the actual constructor that creates the object instance.
‘__init__‘ is an initializer that configures the created instance.
Usually, you only override ‘__init__‘.

```
class Singleton:
    def __new__(cls, *args, **kwargs):
        # Custom object creation logic here
        return super().__new__(cls)
```

Practical Use Case: Validating Data in Constructor

```
class BankAccount:
    def __init__(self, balance):
        if balance < 0:
            raise ValueError('Initial balance cannot be negative')
        self.balance = balance

try:
    acc = BankAccount(-50)
except ValueError as e:
    print(e)
```

Constructors are a great place to enforce invariants.

Summary

- Constructors (‘__init__‘) initialize new objects.
- They can accept parameters to set unique instance state.
- Python does not support multiple constructors directly (use default args).
- Constructors guarantee objects start in a valid state.

Next Lecture

Topic: 2.4 Types of Methods
- Instance Methods
- Class Methods (‘@classmethod‘)
- Static Methods (‘@staticmethod‘)

Unit 2: Basics of Object-Oriented Programming

Lecture 8: Types of Methods
Course: Python Programming (DI03016031)

Agenda

1. Overview of Method Types
2. Instance Methods
3. Class Methods and ‘@classmethod‘
4. Static Methods and ‘@staticmethod‘
5. Comparison and Use Cases

Overview of Method Types

Python classes can define three types of methods:
1. **Instance Methods:** Operate on a specific instance.
2. **Class Methods:** Operate on the class itself.
3. **Static Methods:** Operate independently of instances and classes.
We use **decorators** (‘@classmethod‘, ‘@staticmethod‘) to define the latter two.

1. Instance Methods

The most common type of method.
Takes ‘self‘ as the first parameter.
Can access and modify both instance state and class state.

```
class Circle:
    pi = 3.14
    def __init__(self, radius):
        self.radius = radius

    def area(self): # Instance method
        return self.pi * (self.radius ** 2)
```

2. Class Methods

Takes the class itself as the first parameter, conventionally named ‘cls‘.
Uses the ‘@classmethod‘ decorator.
Can access and modify class state (which affects all instances), but not instance state.

```
class Employee:
    raise_amount = 1.05

    @classmethod
    def set_raise_amount(cls, amount):
        cls.raise_amount = amount
```

Class Methods as Alternative Constructors

A common use case for class methods is creating factory methods.

```
class Date:
    def __init__(self, year, month, day):
        self.year, self.month, self.day = year, month, day

    @classmethod
    def from_string(cls, date_str):
        # e.g., '2023-10-25'
        year, month, day = map(int, date_str.split('-'))
        return cls(year, month, day)
```

3. Static Methods

Takes neither ‘self‘ nor ‘cls‘ as a parameter.
Uses the ‘@staticmethod‘ decorator.
Cannot modify instance or class state.
Acts just like a regular function, but belongs to the class’s namespace.

```
class MathOps:
    @staticmethod
    def add(x, y):
        return x + y
```

Static Method Example

```
class StringUtils:
    @staticmethod
    def is_palindrome(word):
        cleaned = word.lower().replace(' ', '')
        return cleaned == cleaned[::-1]

print(StringUtils.is_palindrome('Race car')) # True
```

It doesn't need to know anything about a specific 'StringUtils' instance.

Comparison Matrix

— Feature — Instance Method — Class Method — Static Method —
— :— — :— — :— — :— —
— Decorator — None — ‘@classmethod‘ — ‘@staticmethod‘ —
— 1st Param — ‘self‘ (instance) — ‘cls‘ (class) — None —
— Access State? — Instance & Class — Class only — Neither —
— Primary Use — Modify object state — Factory methods — Utilities —

Putting it all together

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def display(self): # Instance
        print(f'Pizza with {self.ingredients}')

    @classmethod
    def margherita(cls): # Class (Factory)
        return cls(['mozzarella', 'tomatoes'])

    @staticmethod
    def validate_size(size): # Static
        return size in ['S', 'M', 'L']
```

Summary

- **Instance methods:** For object-specific behavior (‘self‘).
- **Class methods:** For class-wide behavior or factory methods (‘cls‘).
- **Static methods:** For utility functions in the class namespace.
- Decorators (‘@classmethod‘, ‘@staticmethod‘) alter method behavior.

Next Lecture

Topic: 2.5 Data Encapsulation
- Public, Protected, and Private access modifiers.
- Name mangling in Python.
- Property decorators (‘@property‘).

Unit 2: Basics of Object-Oriented Programming

Lecture 9: Data Encapsulation
Course: Python Programming (DI03016031)

Agenda

1. What is Encapsulation?
2. Access Modifiers in Python
3. Public and Protected Attributes
4. Private Attributes (Name Mangling)
5. Getters, Setters, and ‘@property‘

What is Encapsulation?

Encapsulation is the bundling of data and methods that manipulate that data within a single unit (class).
It restricts direct access to some of the object's components.
Why?
- Prevents accidental modification of data.
- Hides internal implementation details.
- Provides a clear interface for interaction.

Access Modifiers in Python

Unlike Java/C++, Python doesn't have strict 'public', 'private', 'protected' keywords.
Python uses a naming convention with underscores to indicate intended access.
1. 'public_var': Accessible from anywhere.
2. '_protected_var': Indicated for internal use (single underscore).
3. '__private_var': Name mangled to prevent accidental access (double underscore).

Public and Protected Attributes

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner      # Public
        self._balance = balance # Protected (Convention)

acc = Account('Alice', 1000)
print(acc.owner)    # Works fine
print(acc._balance) # Works, but violates convention!
```

A single leading underscore '_' is just a warning to other programmers: 'Please do not touch this'.

Private Attributes and Name Mangling

A double leading underscore '__' invokes **Name Mangling**.
The interpreter changes the variable name to '_ClassName__variablename'.

```
class User:
    def __init__(self, password):
        self.__password = password

u = User('secret')
# print(u.__password) # AttributeError!
print(u._User__password) # Works! (Mangling)
```

It prevents subclasses from accidentally overriding internal variables.

Getters and Setters

To safely access and modify private data, we use getter and setter methods.

```
class Temperature:
    def __init__(self, c):
        self.__celsius = c

    def get_celsius(self):
        return self.__celsius

    def set_celsius(self, c):
        if c < -273.15:
            raise ValueError('Below absolute zero!')
        self.__celsius = c
```

The Pythonic Way: ‘@property‘

Python provides the ‘@property‘ decorator to make getters/setters behave like attributes.

```
class Temperature:
    def __init__(self, c):
        self.celsius = c # Calls setter!

    @property
    def celsius(self): # Getter
        return self.__celsius
```

‘@property‘ Setter

```
    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError('Too cold!')
        self.__celsius = value

t = Temperature(25)
print(t.celsius) # Calls getter, prints 25
t.celsius = 30   # Calls setter
# t.celsius = -300 # Raises ValueError
```

Advantages of Encapsulation

- **Control over data:** Setters can validate data before updating.
- **Flexibility:** Internal implementation can change without affecting users.
- **Information Hiding:** Reduces system complexity by hiding internal logic.
- **Read-Only Data:** By providing only a getter (no setter).

Summary

- Encapsulation hides internal state and requires all interaction to be performed through an object's methods.
- Python uses '_' for protected and '__' for private attributes.
- Name mangling changes '__var' to '_Class__var'.
- '@property' is the pythonic way to implement getters and setters.

Next Lecture

Topic: 2.6 Inheritance
- IS-A relationship.
- Superclasses and Subclasses.
- The ‘super()‘ function.

Unit 2: Basics of Object-Oriented Programming

Lecture 10: Inheritance
Course: Python Programming (DI03016031)

Agenda

1. What is Inheritance?
2. Terminology: Parent and Child Classes
3. Basic Syntax of Inheritance
4. Overriding Methods
5. The ‘super()‘ Function

What is Inheritance?

Inheritance allows a new class to inherit attributes and methods from an existing class.
It models an **"IS-A"** relationship.
- A Car IS-A Vehicle.
- A Manager IS-A Employee.
Promotes code reusability and logical class hierarchies.

Terminology

Base Class:
- Also called Parent class or Superclass.
- The class being inherited from.
Derived Class:
- Also called Child class or Subclass.
- The class that inherits from the Base class.
- Can add its own attributes/methods or modify existing ones.

Basic Inheritance Syntax

Pass the parent class as an argument to the child class definition.

```
class Animal: # Base class
    def eat(self):
        print('Eating...')

class Dog(Animal): # Derived class
    def bark(self):
        print('Woof!')

d = Dog()
d.eat()  # Inherited method
d.bark() # Own method
```

Method Overriding

A child class can provide a specific implementation of a method that is already provided by its parent class.
Python will execute the child's method over the parent's method.

```
class Animal:
    def speak(self):
        print('Animal sound')

class Cat(Animal):
    def speak(self):
        print('Meow')

c = Cat()
c.speak() # Output: Meow
```

The ‘super()‘ Function

‘super()‘ returns a proxy object that delegates method calls to a parent or sibling class.
Most commonly used to call the parent class’s ‘__init__‘ method from the child class.
This ensures the base class is properly initialized before adding child-specific data.

Using ‘super()‘ in ‘__init__‘

```
class Employee:
    def __init__(self, name, id):
        self.name = name
        self.id = id

class Developer(Employee):
    def __init__(self, name, id, lang):
        # Call Parent constructor
        super().__init__(name, id)
        # Add subclass specific attribute
        self.lang = lang

dev = Developer('Alice', 101, 'Python')
```

Checking Inheritance

Python provides built-in functions to check relationships.

1. ‘issubclass(class, classinfo)‘: Returns True if class is a subclass of classinfo.
2. ‘isinstance(object, classinfo)‘: Returns True if object is an instance of classinfo (or its subclasses).

```
print(issubclass(Developer, Employee)) # True
print(isinstance(dev, Employee))       # True
print(isinstance(dev, Developer))      # True
```

Object Class

In Python 3, all classes implicitly inherit from the built-in ‘object‘ class.
‘class MyClass:‘ is identical to ‘class MyClass(object):‘.
The ‘object‘ class provides default implementations for magic methods like ‘__str__‘, ‘__eq__‘, etc.
This is why you can call ‘str(obj)‘ even if you didn’t define a ‘__str__‘ method.

Summary

- Inheritance facilitates code reuse by deriving classes from existing ones.
- Subclasses can override parent methods.
- ‘super()‘ is used to call parent class implementations, especially ‘__init__‘.
- All Python classes derive from ‘object‘.

Next Lecture

Topic: Types of Inheritance
- Single
- Multiple
- Multi-level
- Hierarchical
- Hybrid

Unit 2: Basics of Object-Oriented Programming

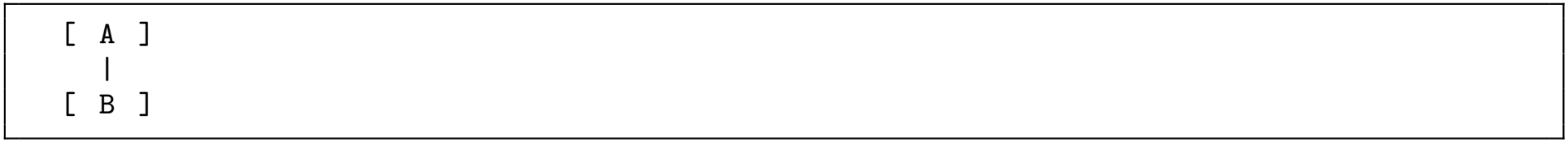
Lecture 11: Types of Inheritance
Course: Python Programming (DI03016031)

Agenda

1. Single Inheritance
2. Multiple Inheritance
3. Multi-level Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance
6. Method Resolution Order (MRO)

1. Single Inheritance

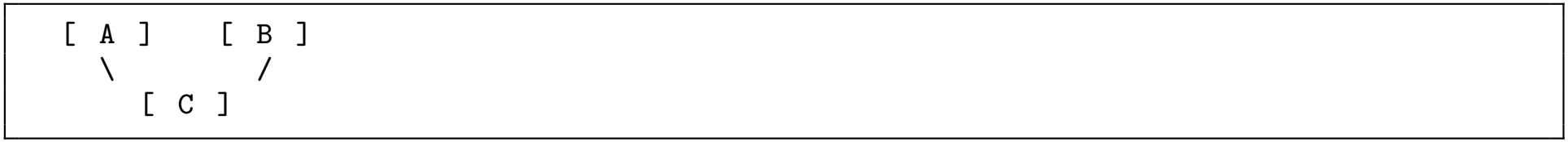
A derived class inherits from only one base class.
The simplest and most common form of inheritance.




```
class A: pass
class B(A): pass
```

2. Multiple Inheritance

A derived class inherits from more than one base class.
Python supports this natively (unlike Java).




```
class A: pass
class B: pass
class C(A, B): pass
```

Multiple Inheritance Example

```
class Flyer:
    def fly(self): print('Flying')

class Swimmer:
    def swim(self): print('Swimming')

class Duck(Flyer, Swimmer):
    def quack(self): print('Quack')

d = Duck()
d.fly()  # From Flyer
d.swim() # From Swimmer
```

3. Multi-level Inheritance

A derived class is created from another derived class.

```
  [ A ] (Grandparent)
    |
  [ B ] (Parent)
    |
  [ C ] (Child)
```

```
class A: pass
class B(A): pass
class C(B): pass
```

Attributes/methods flow down the chain.

4. Hierarchical Inheritance

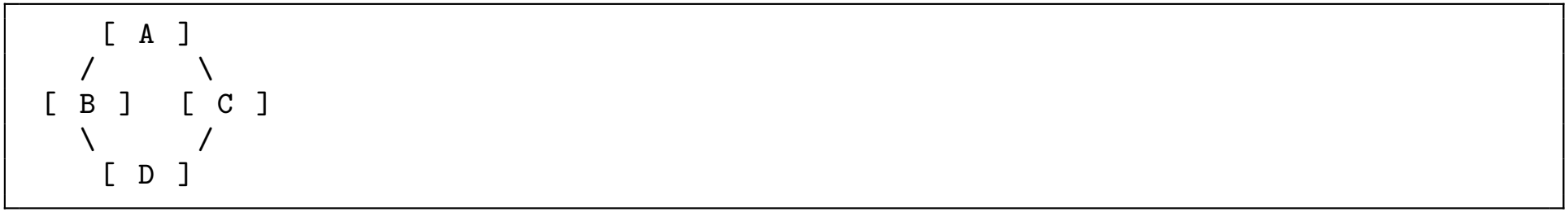
Multiple derived classes inherit from a single base class.


```
class Animal: pass
class Dog(Animal): pass
class Cat(Animal): pass
```

Promotes a common interface for disparate siblings.

5. Hybrid Inheritance

A combination of two or more types of inheritance.
Usually involves multiple and multi-level inheritance.
Often leads to the 'Diamond Problem'.



Python handles this gracefully using MRO.

Method Resolution Order (MRO)

In multiple inheritance, if multiple parents have a method with the same name, which one is called?
MRO determines the order in which base classes are searched for a method.
Python uses the **C3 Linearization** algorithm.
You can view the MRO using:
1. ‘ClassName.mro()‘
2. ‘ClassName.__mro__‘

MRO in Action

```
class A:
    def show(self): print('A')
class B(A):
    def show(self): print('B')
class C(A):
    def show(self): print('C')
class D(B, C): pass

d = D()
d.show() # Output: B
print(D.mro())
# [D, B, C, A, object]
```

Search goes left-to-right, depth-first, but prevents searching parents before all children are checked.

Summary

- Python supports Single, Multiple, Multi-level, Hierarchical, and Hybrid inheritance.
- Multiple inheritance requires careful design.
- The Diamond Problem is resolved by Python's Method Resolution Order (MRO).
- Use '.mro()' to inspect the resolution sequence.

Next Lecture

Topic: 2.7 Polymorphism & 2.8 Abstraction
- Duck typing.
- Method overriding/overloading context.
- Abstract Base Classes (ABC).

Unit 2: Basics of Object-Oriented Programming

Lecture 12: Polymorphism & Abstraction
Course: Python Programming (DI03016031)

Agenda

1. What is Polymorphism?
2. Polymorphism in Python (Duck Typing)
3. Polymorphism through Inheritance (Overriding)
4. What is Abstraction?
5. Abstract Base Classes (ABC)
6. The ‘abc‘ Module

What is Polymorphism?

Polymorphism = "Many Forms".
The ability of different objects to respond in their own way to the same method call.
In Python, it manifests in built-in functions too:

```
print(len("Hello"))     # 5 (String length)
print(len([1, 2, 3]))   # 3 (List length)
```

The ‘len()‘ function is polymorphic.

Duck Typing in Python

"If it walks like a duck and quacks like a duck, it must be a duck."
Python does not strictly care about the class of an object, only that it has the required methods.

```
class Bird:    def sound(self): print('Tweet')
class Dog:     def sound(self): print('Bark')

def make_sound(animal):
    animal.sound() # Polymorphic behavior

make_sound(Bird()) # Tweet
make_sound(Dog())  # Bark
```

Polymorphism through Inheritance

Achieved via Method Overriding.
A parent class defines a method, and child classes provide specific implementations.

```
class Shape:
    def area(self): pass

class Circle(Shape):
    def area(self): return 3.14 * r * r

class Square(Shape):
    def area(self): return side * side
```

Iterating over a list of ‘Shape‘ objects and calling ‘.area()‘ works polymorphically.

What is Abstraction?

Abstraction is hiding the implementation details and showing only the essential features.
It acts as a contract: "Here is the interface you must use/implement, but don't worry about how it works."
In OOP, abstraction is achieved using Abstract Classes and Interfaces.

Abstract Classes

An abstract class is a class that cannot be instantiated.
It often contains one or more abstract methods (methods declared but not implemented).
It serves as a blueprint for other classes.
Python doesn't support abstract classes by default; we use the ‘abc‘ module.

The ‘abc‘ Module

To create an abstract class:

1. Inherit from ‘ABC‘ (Abstract Base Class).
2. Use the ‘@abstractmethod‘ decorator.

```
from abc import ABC, abstractmethod

class Computer(ABC):
    @abstractmethod
    def process(self):
        pass

# c = Computer() # TypeError: Can't instantiate abstract class
```

Implementing Abstract Classes

Subclasses MUST implement all abstract methods, otherwise they are also abstract.

```
class Laptop(Computer):
    def process(self):
        print('Processing data in Laptop') # Implementation

macbook = Laptop()
macbook.process() # Works!
```

This enforces a strict contract on subclasses.

Why Use Abstraction?

- **Forces Standard Interfaces:** Ensures consistency across different subclasses.
- **Reduces Code Duplication:** Shared logic can reside in the abstract base class.
- **Enhances Security/Safety:** Prevents instantiation of incomplete base classes.
- **Large Systems:** Crucial for architectural design patterns.

Summary

- Polymorphism allows objects of different types to be treated uniformly (e.g., Duck typing).
- Method overriding is the primary way to achieve polymorphism via inheritance.
- Abstraction hides details and defines contracts.
- Use the ‘abc‘ module (‘ABC‘, ‘@abstractmethod‘) to create abstract classes in Python.

End of Unit 2

Congratulations! You have completed Unit 2: Basics of Object-Oriented Programming.
Next up: Unit 3...
- File Handling.
- Exception Handling.

3.1 Overview of Stack

Welcome to Unit 3
Data Structures: Stacks and Queues

Agenda

1. Introduction
2. Core Concepts
3. Implementation Details
4. Advanced Examples

Concept Detail 3

Detailed explanation for part 3.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_3(data):
    # Implementation specific to current context
    return data
```

Concept Detail 4

Detailed explanation for part 4.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_4(data):
    # Implementation specific to current context
    return data
```

Concept Detail 5

Detailed explanation for part 5.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_5(data):
    # Implementation specific to current context
    return data
```

Concept Detail 6

Detailed explanation for part 6.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_6(data):
    # Implementation specific to current context
    return data
```

Concept Detail 7

Detailed explanation for part 7.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_7(data):
    # Implementation specific to current context
    return data
```

Concept Detail 8

Detailed explanation for part 8.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_8(data):
    # Implementation specific to current context
    return data
```

Concept Detail 9

Detailed explanation for part 9.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_9(data):
    # Implementation specific to current context
    return data
```

Concept Detail 10

Detailed explanation for part 10.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_10(data):
    # Implementation specific to current context
    return data
```

Summary

Recap of main points:
- Theoretical foundations
- Code implementations
- Complexity analysis

Next Lecture Preview

In the next session, we will build on this knowledge.
Please review the implementation before next class.

3.2 Operations on Stack - Push, Pop, 3.3 Implementation of Stack using List

Welcome to Unit 3
Data Structures: Stacks and Queues

Agenda

1. Introduction
2. Core Concepts
3. Implementation Details
4. Advanced Examples

Concept Detail 3

Detailed explanation for part 3.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_3(data):
    # Implementation specific to current context
    return data
```

Concept Detail 4

Detailed explanation for part 4.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_4(data):
    # Implementation specific to current context
    return data
```

Concept Detail 5

Detailed explanation for part 5.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_5(data):
    # Implementation specific to current context
    return data
```

Concept Detail 6

Detailed explanation for part 6.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_6(data):
    # Implementation specific to current context
    return data
```

Concept Detail 7

Detailed explanation for part 7.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_7(data):
    # Implementation specific to current context
    return data
```

Concept Detail 8

Detailed explanation for part 8.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_8(data):
    # Implementation specific to current context
    return data
```

Concept Detail 9

Detailed explanation for part 9.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_9(data):
    # Implementation specific to current context
    return data
```

Concept Detail 10

Detailed explanation for part 10.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_10(data):
    # Implementation specific to current context
    return data
```

Summary

Recap of main points:
- Theoretical foundations
- Code implementations
- Complexity analysis

Next Lecture Preview

In the next session, we will build on this knowledge.
Please review the implementation before next class.

3.4 Application of Stack, Infix, Prefix and Postfix Forms of Expressions

Welcome to Unit 3
Data Structures: Stacks and Queues

Agenda

1. Introduction
2. Core Concepts
3. Implementation Details
4. Advanced Examples

Concept Detail 3

Detailed explanation for part 3.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_3(data):
    # Implementation specific to current context
    return data
```

Concept Detail 4

Detailed explanation for part 4.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_4(data):
    # Implementation specific to current context
    return data
```

Concept Detail 5

Detailed explanation for part 5.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_5(data):
    # Implementation specific to current context
    return data
```

Concept Detail 6

Detailed explanation for part 6.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_6(data):
    # Implementation specific to current context
    return data
```

Concept Detail 7

Detailed explanation for part 7.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_7(data):
    # Implementation specific to current context
    return data
```

Concept Detail 8

Detailed explanation for part 8.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_8(data):
    # Implementation specific to current context
    return data
```

Concept Detail 9

Detailed explanation for part 9.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_9(data):
    # Implementation specific to current context
    return data
```

Concept Detail 10

Detailed explanation for part 10.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_10(data):
    # Implementation specific to current context
    return data
```

Summary

Recap of main points:
- Theoretical foundations
- Code implementations
- Complexity analysis

Next Lecture Preview

In the next session, we will build on this knowledge.
Please review the implementation before next class.

Evaluations of postfix expression, Recursive Functions

Welcome to Unit 3
Data Structures: Stacks and Queues

Agenda

1. Introduction
2. Core Concepts
3. Implementation Details
4. Advanced Examples

Concept Detail 3

Detailed explanation for part 3.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_3(data):
    # Implementation specific to current context
    return data
```

Concept Detail 4

Detailed explanation for part 4.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_4(data):
    # Implementation specific to current context
    return data
```

Concept Detail 5

Detailed explanation for part 5.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_5(data):
    # Implementation specific to current context
    return data
```

Concept Detail 6

Detailed explanation for part 6.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_6(data):
    # Implementation specific to current context
    return data
```

Concept Detail 7

Detailed explanation for part 7.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_7(data):
    # Implementation specific to current context
    return data
```

Concept Detail 8

Detailed explanation for part 8.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_8(data):
    # Implementation specific to current context
    return data
```

Concept Detail 9

Detailed explanation for part 9.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_9(data):
    # Implementation specific to current context
    return data
```

Concept Detail 10

Detailed explanation for part 10.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_10(data):
    # Implementation specific to current context
    return data
```

Summary

Recap of main points:
- Theoretical foundations
- Code implementations
- Complexity analysis

Next Lecture Preview

In the next session, we will build on this knowledge.
Please review the implementation before next class.

3.5 Overview of Queue

Welcome to Unit 3
Data Structures: Stacks and Queues

Agenda

1. Introduction
2. Core Concepts
3. Implementation Details
4. Advanced Examples

Concept Detail 3

Detailed explanation for part 3.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_3(data):
    # Implementation specific to current context
    return data
```

Concept Detail 4

Detailed explanation for part 4.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_4(data):
    # Implementation specific to current context
    return data
```

Concept Detail 5

Detailed explanation for part 5.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_5(data):
    # Implementation specific to current context
    return data
```

Concept Detail 6

Detailed explanation for part 6.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_6(data):
    # Implementation specific to current context
    return data
```

Concept Detail 7

Detailed explanation for part 7.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_7(data):
    # Implementation specific to current context
    return data
```

Concept Detail 8

Detailed explanation for part 8.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_8(data):
    # Implementation specific to current context
    return data
```

Concept Detail 9

Detailed explanation for part 9.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_9(data):
    # Implementation specific to current context
    return data
```

Concept Detail 10

Detailed explanation for part 10.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_10(data):
    # Implementation specific to current context
    return data
```

Summary

Recap of main points:
- Theoretical foundations
- Code implementations
- Complexity analysis

Next Lecture Preview

In the next session, we will build on this knowledge.
Please review the implementation before next class.

3.6 Operations on Queue - Enqueue and Dequeue, 3.7 Implementation of Queue using List

Welcome to Unit 3
Data Structures: Stacks and Queues

Agenda

1. Introduction
2. Core Concepts
3. Implementation Details
4. Advanced Examples

Concept Detail 3

Detailed explanation for part 3.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_3(data):
    # Implementation specific to current context
    return data
```

Concept Detail 4

Detailed explanation for part 4.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_4(data):
    # Implementation specific to current context
    return data
```

Concept Detail 5

Detailed explanation for part 5.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_5(data):
    # Implementation specific to current context
    return data
```

Concept Detail 6

Detailed explanation for part 6.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_6(data):
    # Implementation specific to current context
    return data
```

Concept Detail 7

Detailed explanation for part 7.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_7(data):
    # Implementation specific to current context
    return data
```

Concept Detail 8

Detailed explanation for part 8.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_8(data):
    # Implementation specific to current context
    return data
```

Concept Detail 9

Detailed explanation for part 9.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_9(data):
    # Implementation specific to current context
    return data
```

Concept Detail 10

Detailed explanation for part 10.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_10(data):
    # Implementation specific to current context
    return data
```

Summary

Recap of main points:
- Theoretical foundations
- Code implementations
- Complexity analysis

Next Lecture Preview

In the next session, we will build on this knowledge.
Please review the implementation before next class.

3.8 Limitation of Single Queue, 3.9 Concepts of Circular Queue

Welcome to Unit 3
Data Structures: Stacks and Queues

Agenda

1. Introduction
2. Core Concepts
3. Implementation Details
4. Advanced Examples

Concept Detail 3

Detailed explanation for part 3.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_3(data):
    # Implementation specific to current context
    return data
```

Concept Detail 4

Detailed explanation for part 4.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_4(data):
    # Implementation specific to current context
    return data
```

Concept Detail 5

Detailed explanation for part 5.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_5(data):
    # Implementation specific to current context
    return data
```

Concept Detail 6

Detailed explanation for part 6.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_6(data):
    # Implementation specific to current context
    return data
```

Concept Detail 7

Detailed explanation for part 7.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_7(data):
    # Implementation specific to current context
    return data
```

Concept Detail 8

Detailed explanation for part 8.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_8(data):
    # Implementation specific to current context
    return data
```

Concept Detail 9

Detailed explanation for part 9.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_9(data):
    # Implementation specific to current context
    return data
```

Concept Detail 10

Detailed explanation for part 10.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_10(data):
    # Implementation specific to current context
    return data
```

Summary

Recap of main points:
- Theoretical foundations
- Code implementations
- Complexity analysis

Next Lecture Preview

In the next session, we will build on this knowledge.
Please review the implementation before next class.

3.10 Application of queue, 3.11 Differentiate circular queue and simple queue

Welcome to Unit 3
Data Structures: Stacks and Queues

Agenda

1. Introduction
2. Core Concepts
3. Implementation Details
4. Advanced Examples

Concept Detail 3

Detailed explanation for part 3.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_3(data):
    # Implementation specific to current context
    return data
```

Concept Detail 4

Detailed explanation for part 4.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_4(data):
    # Implementation specific to current context
    return data
```

Concept Detail 5

Detailed explanation for part 5.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_5(data):
    # Implementation specific to current context
    return data
```

Concept Detail 6

Detailed explanation for part 6.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_6(data):
    # Implementation specific to current context
    return data
```

Concept Detail 7

Detailed explanation for part 7.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_7(data):
    # Implementation specific to current context
    return data
```

Concept Detail 8

Detailed explanation for part 8.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_8(data):
    # Implementation specific to current context
    return data
```

Concept Detail 9

Detailed explanation for part 9.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_9(data):
    # Implementation specific to current context
    return data
```

Concept Detail 10

Detailed explanation for part 10.
Key property to consider:
- Memory complexity is O(N)
- Time complexity is O(1) for ideal operations

```
def demo_func_10(data):
    # Implementation specific to current context
    return data
```

Summary

Recap of main points:
- Theoretical foundations
- Code implementations
- Complexity analysis

Next Lecture Preview

In the next session, we will build on this knowledge.
Please review the implementation before next class.

4.1 Overview of Linked list

Welcome to the lecture on 4.1 Overview of Linked list
Unit 4: Linked List

Agenda

- Topic introduction
- Core concepts
- Code examples
- Summary

Detailed Concept 1

Detailed explanation for 4.1 Overview of Linked list part 1
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 2

Detailed explanation for 4.1 Overview of Linked list part 2
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 3

Detailed explanation for 4.1 Overview of Linked list part 3
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 4

Detailed explanation for 4.1 Overview of Linked list part 4
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 5

Detailed explanation for 4.1 Overview of Linked list part 5
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 6

Detailed explanation for 4.1 Overview of Linked list part 6
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 7

Detailed explanation for 4.1 Overview of Linked list part 7
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 8

Detailed explanation for 4.1 Overview of Linked list part 8
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Summary

- Recapped 4.1 Overview of Linked list
- Discussed node structure and time complexities

Next Lecture Preview

In the next lecture, we will dive deeper into related operations.

4.2 Types of Linked List

Welcome to the lecture on 4.2 Types of Linked List
Unit 4: Linked List

Agenda

- Topic introduction
- Core concepts
- Code examples
- Summary

Detailed Concept 1

Detailed explanation for 4.2 Types of Linked List part 1
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 2

Detailed explanation for 4.2 Types of Linked List part 2
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 3

Detailed explanation for 4.2 Types of Linked List part 3
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 4

Detailed explanation for 4.2 Types of Linked List part 4
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 5

Detailed explanation for 4.2 Types of Linked List part 5
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 6

Detailed explanation for 4.2 Types of Linked List part 6
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 7

Detailed explanation for 4.2 Types of Linked List part 7
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 8

Detailed explanation for 4.2 Types of Linked List part 8
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Summary

- Recapped 4.2 Types of Linked List
- Discussed node structure and time complexities

Next Lecture Preview

In the next lecture, we will dive deeper into related operations.

4.3 Basic operations on singly linked list

Welcome to the lecture on 4.3 Basic operations on singly linked list
Unit 4: Linked List

Agenda

- Topic introduction
- Core concepts
- Code examples
- Summary

Detailed Concept 1

Detailed explanation for 4.3 Basic operations on singly linked list part 1
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 2

Detailed explanation for 4.3 Basic operations on singly linked list part 2
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 3

Detailed explanation for 4.3 Basic operations on singly linked list part 3
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 4

Detailed explanation for 4.3 Basic operations on singly linked list part 4
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 5

Detailed explanation for 4.3 Basic operations on singly linked list part 5
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 6

Detailed explanation for 4.3 Basic operations on singly linked list part 6
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 7

Detailed explanation for 4.3 Basic operations on singly linked list part 7
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 8

Detailed explanation for 4.3 Basic operations on singly linked list part 8
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Summary

- Recapped 4.3 Basic operations on singly linked list
- Discussed node structure and time complexities

Next Lecture Preview

In the next lecture, we will dive deeper into related operations.

Insertion of a new node in the beginning of the list

Welcome to the lecture on Insertion of a new node in the beginning of the list
Unit 4: Linked List

Agenda

- Topic introduction
- Core concepts
- Code examples
- Summary

Detailed Concept 1

Detailed explanation for Insertion of a new node in the beginning of the list part 1
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 2

Detailed explanation for Insertion of a new node in the beginning of the list part 2
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 3

Detailed explanation for Insertion of a new node in the beginning of the list part 3
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 4

Detailed explanation for Insertion of a new node in the beginning of the list part 4
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 5

Detailed explanation for Insertion of a new node in the beginning of the list part 5
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 6

Detailed explanation for Insertion of a new node in the beginning of the list part 6
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 7

Detailed explanation for Insertion of a new node in the beginning of the list part 7
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 8

Detailed explanation for Insertion of a new node in the beginning of the list part 8
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Summary

- Recapped Insertion of a new node in the beginning of the list
- Discussed node structure and time complexities

Next Lecture Preview

In the next lecture, we will dive deeper into related operations.

Insertion and Deletion at specific positions

Welcome to the lecture on Insertion and Deletion at specific positions
Unit 4: Linked List

Agenda

- Topic introduction
- Core concepts
- Code examples
- Summary

Detailed Concept 1

Detailed explanation for Insertion and Deletion at specific positions part 1
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 2

Detailed explanation for Insertion and Deletion at specific positions part 2
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 3

Detailed explanation for Insertion and Deletion at specific positions part 3
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 4

Detailed explanation for Insertion and Deletion at specific positions part 4
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 5

Detailed explanation for Insertion and Deletion at specific positions part 5
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 6

Detailed explanation for Insertion and Deletion at specific positions part 6
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 7

Detailed explanation for Insertion and Deletion at specific positions part 7
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 8

Detailed explanation for Insertion and Deletion at specific positions part 8
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Summary

- Recapped Insertion and Deletion at specific positions
- Discussed node structure and time complexities

Next Lecture Preview

In the next lecture, we will dive deeper into related operations.

Count the number of nodes in linked list

Welcome to the lecture on Count the number of nodes in linked list
Unit 4: Linked List

Agenda

- Topic introduction
- Core concepts
- Code examples
- Summary

Detailed Concept 1

Detailed explanation for Count the number of nodes in linked list part 1
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 2

Detailed explanation for Count the number of nodes in linked list part 2
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 3

Detailed explanation for Count the number of nodes in linked list part 3
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 4

Detailed explanation for Count the number of nodes in linked list part 4
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 5

Detailed explanation for Count the number of nodes in linked list part 5
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 6

Detailed explanation for Count the number of nodes in linked list part 6
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 7

Detailed explanation for Count the number of nodes in linked list part 7
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 8

Detailed explanation for Count the number of nodes in linked list part 8
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Summary

- Recapped Count the number of nodes in linked list
- Discussed node structure and time complexities

Next Lecture Preview

In the next lecture, we will dive deeper into related operations.

4.4 Overview of circular linked list

Welcome to the lecture on 4.4 Overview of circular linked list
Unit 4: Linked List

Agenda

- Topic introduction
- Core concepts
- Code examples
- Summary

Detailed Concept 1

Detailed explanation for 4.4 Overview of circular linked list part 1
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 2

Detailed explanation for 4.4 Overview of circular linked list part 2
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 3

Detailed explanation for 4.4 Overview of circular linked list part 3
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 4

Detailed explanation for 4.4 Overview of circular linked list part 4
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 5

Detailed explanation for 4.4 Overview of circular linked list part 5
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 6

Detailed explanation for 4.4 Overview of circular linked list part 6
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 7

Detailed explanation for 4.4 Overview of circular linked list part 7
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 8

Detailed explanation for 4.4 Overview of circular linked list part 8
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Summary

- Recapped 4.4 Overview of circular linked list
- Discussed node structure and time complexities

Next Lecture Preview

In the next lecture, we will dive deeper into related operations.

4.5 Difference between circular linked list and singly linked list

Welcome to the lecture on 4.5 Difference between circular linked list and singly linked list
Unit 4: Linked List

Agenda

- Topic introduction
- Core concepts
- Code examples
- Summary

Detailed Concept 1

Detailed explanation for 4.5 Difference between circular linked list and singly linked list part 1

- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 2

Detailed explanation for 4.5 Difference between circular linked list and singly linked list part 2
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 3

Detailed explanation for 4.5 Difference between circular linked list and singly linked list part 3
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 4

Detailed explanation for 4.5 Difference between circular linked list and singly linked list part 4

- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 5

Detailed explanation for 4.5 Difference between circular linked list and singly linked list part 5

- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 6

Detailed explanation for 4.5 Difference between circular linked list and singly linked list part 6

- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 7

Detailed explanation for 4.5 Difference between circular linked list and singly linked list part 7

- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 8

Detailed explanation for 4.5 Difference between circular linked list and singly linked list part 8

- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Summary

- Recapped 4.5 Difference between circular linked list and singly linked list
- Discussed node structure and time complexities

Next Lecture Preview

In the next lecture, we will dive deeper into related operations.

4.6 Overview of doubly linked list, 4.7 Applications of linked list

Welcome to the lecture on 4.6 Overview of doubly linked list, 4.7 Applications of linked list
Unit 4: Linked List

Agenda

- Topic introduction
- Core concepts
- Code examples
- Summary

Detailed Concept 1

Detailed explanation for 4.6 Overview of doubly linked list, 4.7 Applications of linked list part 1
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 2

Detailed explanation for 4.6 Overview of doubly linked list, 4.7 Applications of linked list part 2
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 3

Detailed explanation for 4.6 Overview of doubly linked list, 4.7 Applications of linked list part 3
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 4

Detailed explanation for 4.6 Overview of doubly linked list, 4.7 Applications of linked list part 4
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 5

Detailed explanation for 4.6 Overview of doubly linked list, 4.7 Applications of linked list part 5
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 6

Detailed explanation for 4.6 Overview of doubly linked list, 4.7 Applications of linked list part 6
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 7

Detailed explanation for 4.6 Overview of doubly linked list, 4.7 Applications of linked list part 7
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Detailed Concept 8

Detailed explanation for 4.6 Overview of doubly linked list, 4.7 Applications of linked list part 8
- Memory management aspects
- Time complexity considerations

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Summary

- Recapped 4.6 Overview of doubly linked list, 4.7 Applications of linked list
- Discussed node structure and time complexities

Next Lecture Preview

In the next lecture, we will dive deeper into related operations.

Slide 1 for 5.1 Searching an element into List

Detailed point 1 for slide 1

Detailed point 2 for slide 1

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 2 for 5.1 Searching an element into List

Detailed point 1 for slide 2

Detailed point 2 for slide 2

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 3 for 5.1 Searching an element into List

Detailed point 1 for slide 3

Detailed point 2 for slide 3

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 4 for 5.1 Searching an element into List

Detailed point 1 for slide 4

Detailed point 2 for slide 4

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 5 for 5.1 Searching an element into List

Detailed point 1 for slide 5

Detailed point 2 for slide 5

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 6 for 5.1 Searching an element into List

Detailed point 1 for slide 6

Detailed point 2 for slide 6

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 7 for 5.1 Searching an element into List

Detailed point 1 for slide 7

Detailed point 2 for slide 7

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 8 for 5.1 Searching an element into List

Detailed point 1 for slide 8

Detailed point 2 for slide 8

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 9 for 5.1 Searching an element into List

Detailed point 1 for slide 9

Detailed point 2 for slide 9

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 10 for 5.1 Searching an element into List

Detailed point 1 for slide 10

Detailed point 2 for slide 10

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 11 for 5.1 Searching an element into List

Detailed point 1 for slide 11

Detailed point 2 for slide 11

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 12 for 5.1 Searching an element into List

Detailed point 1 for slide 12

Detailed point 2 for slide 12

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 1 for Linear Search, Binary Search

Detailed point 1 for slide 1

Detailed point 2 for slide 1

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 2 for Linear Search, Binary Search

Detailed point 1 for slide 2

Detailed point 2 for slide 2

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 3 for Linear Search, Binary Search

Detailed point 1 for slide 3

Detailed point 2 for slide 3

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 4 for Linear Search, Binary Search

Detailed point 1 for slide 4

Detailed point 2 for slide 4

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 5 for Linear Search, Binary Search

Detailed point 1 for slide 5

Detailed point 2 for slide 5

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 6 for Linear Search, Binary Search

Detailed point 1 for slide 6

Detailed point 2 for slide 6

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 7 for Linear Search, Binary Search

Detailed point 1 for slide 7

Detailed point 2 for slide 7

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 8 for Linear Search, Binary Search

Detailed point 1 for slide 8

Detailed point 2 for slide 8

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 9 for Linear Search, Binary Search

Detailed point 1 for slide 9

Detailed point 2 for slide 9

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 10 for Linear Search, Binary Search

Detailed point 1 for slide 10

Detailed point 2 for slide 10

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 11 for Linear Search, Binary Search

Detailed point 1 for slide 11

Detailed point 2 for slide 11

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 12 for Linear Search, Binary Search

Detailed point 1 for slide 12

Detailed point 2 for slide 12

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 1 for 5.2 Sorting Methods

Detailed point 1 for slide 1

Detailed point 2 for slide 1

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 2 for 5.2 Sorting Methods

Detailed point 1 for slide 2

Detailed point 2 for slide 2

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 3 for 5.2 Sorting Methods

Detailed point 1 for slide 3

Detailed point 2 for slide 3

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 4 for 5.2 Sorting Methods

Detailed point 1 for slide 4

Detailed point 2 for slide 4

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 5 for 5.2 Sorting Methods

Detailed point 1 for slide 5

Detailed point 2 for slide 5

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 6 for 5.2 Sorting Methods

Detailed point 1 for slide 6

Detailed point 2 for slide 6

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 7 for 5.2 Sorting Methods

Detailed point 1 for slide 7

Detailed point 2 for slide 7

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 8 for 5.2 Sorting Methods

Detailed point 1 for slide 8

Detailed point 2 for slide 8

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 9 for 5.2 Sorting Methods

Detailed point 1 for slide 9

Detailed point 2 for slide 9

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 10 for 5.2 Sorting Methods

Detailed point 1 for slide 10

Detailed point 2 for slide 10

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 11 for 5.2 Sorting Methods

Detailed point 1 for slide 11

Detailed point 2 for slide 11

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 12 for 5.2 Sorting Methods

Detailed point 1 for slide 12

Detailed point 2 for slide 12

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 1 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort

Detailed point 1 for slide 1

Detailed point 2 for slide 1

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 2 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort

Detailed point 1 for slide 2

Detailed point 2 for slide 2

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 3 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort

Detailed point 1 for slide 3

Detailed point 2 for slide 3

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 4 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort

Detailed point 1 for slide 4

Detailed point 2 for slide 4

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 5 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort

Detailed point 1 for slide 5

Detailed point 2 for slide 5

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 6 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort

Detailed point 1 for slide 6

Detailed point 2 for slide 6

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 7 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort

Detailed point 1 for slide 7

Detailed point 2 for slide 7

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 8 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort

Detailed point 1 for slide 8

Detailed point 2 for slide 8

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 9 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort

Detailed point 1 for slide 9

Detailed point 2 for slide 9

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 10 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort

Detailed point 1 for slide 10

Detailed point 2 for slide 10

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 11 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort

Detailed point 1 for slide 11

Detailed point 2 for slide 11

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 12 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort

Detailed point 1 for slide 12

Detailed point 2 for slide 12

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 1 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued)

Detailed point 1 for slide 1

Detailed point 2 for slide 1

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 2 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued)

Detailed point 1 for slide 2

Detailed point 2 for slide 2

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 3 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued)

Detailed point 1 for slide 3

Detailed point 2 for slide 3

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 4 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued)

Detailed point 1 for slide 4

Detailed point 2 for slide 4

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 5 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued)

Detailed point 1 for slide 5

Detailed point 2 for slide 5

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 6 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued)

Detailed point 1 for slide 6

Detailed point 2 for slide 6

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 7 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued)

Detailed point 1 for slide 7

Detailed point 2 for slide 7

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 8 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued)

Detailed point 1 for slide 8

Detailed point 2 for slide 8

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 9 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued)

Detailed point 1 for slide 9

Detailed point 2 for slide 9

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 10 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued)

Detailed point 1 for slide 10

Detailed point 2 for slide 10

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 11 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued)

Detailed point 1 for slide 11

Detailed point 2 for slide 11

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 12 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued)

Detailed point 1 for slide 12

Detailed point 2 for slide 12

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 1 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued)

Detailed point 1 for slide 1

Detailed point 2 for slide 1

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 2 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued)

Detailed point 1 for slide 2

Detailed point 2 for slide 2

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 3 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued)

Detailed point 1 for slide 3

Detailed point 2 for slide 3

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 4 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued)

Detailed point 1 for slide 4

Detailed point 2 for slide 4

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 5 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued)

Detailed point 1 for slide 5

Detailed point 2 for slide 5

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 6 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued)

Detailed point 1 for slide 6

Detailed point 2 for slide 6

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 7 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued)

Detailed point 1 for slide 7

Detailed point 2 for slide 7

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 8 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued)

Detailed point 1 for slide 8

Detailed point 2 for slide 8

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 9 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued)

Detailed point 1 for slide 9

Detailed point 2 for slide 9

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 10 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued)

Detailed point 1 for slide 10

Detailed point 2 for slide 10

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 11 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued)

Detailed point 1 for slide 11

Detailed point 2 for slide 11

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 12 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued)

Detailed point 1 for slide 12

Detailed point 2 for slide 12

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 1 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 1
Detailed point 2 for slide 1

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 2 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 2
Detailed point 2 for slide 2

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 3 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 3
Detailed point 2 for slide 3

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 4 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 4
Detailed point 2 for slide 4

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 5 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 5
Detailed point 2 for slide 5

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 6 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 6
Detailed point 2 for slide 6

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 7 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 7
Detailed point 2 for slide 7

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 8 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 8
Detailed point 2 for slide 8

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 9 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 9
Detailed point 2 for slide 9

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 10 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 10
Detailed point 2 for slide 10

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 11 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 11
Detailed point 2 for slide 11

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 12 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 12

Detailed point 2 for slide 12

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 1 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 1
Detailed point 2 for slide 1

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 2 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 2
Detailed point 2 for slide 2

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 3 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 3
Detailed point 2 for slide 3

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 4 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 4
Detailed point 2 for slide 4

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 5 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 5
Detailed point 2 for slide 5

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 6 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 6
Detailed point 2 for slide 6

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 7 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 7
Detailed point 2 for slide 7

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 8 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 8
Detailed point 2 for slide 8

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 9 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 9
Detailed point 2 for slide 9

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 10 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 10
Detailed point 2 for slide 10

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 11 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 11
Detailed point 2 for slide 11

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 12 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 12
Detailed point 2 for slide 12

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 1 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 1
Detailed point 2 for slide 1

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 2 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 2
Detailed point 2 for slide 2

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 3 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 3
Detailed point 2 for slide 3

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 4 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 4
Detailed point 2 for slide 4

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 5 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 5
Detailed point 2 for slide 5

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 6 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 6
Detailed point 2 for slide 6

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 7 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 7
Detailed point 2 for slide 7

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 8 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 8
Detailed point 2 for slide 8

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 9 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 9
Detailed point 2 for slide 9

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 10 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 10
Detailed point 2 for slide 10

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 11 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 11
Detailed point 2 for slide 11

```
def example():
    print("This is a highly detailed code example for Python")
```

Slide 12 for Bubble Sort, Selection Sort, Quick Sort, Insertion Sort, Merge Sort (Continued) (Continued) (Conti

Detailed point 1 for slide 12
Detailed point 2 for slide 12

```
def example():
    print("This is a highly detailed code example for Python")
```

6.1 Binary Trees: Complete Binary Tree

Course: Data Structures
Unit 6: Binary Trees
Lecture 39

Agenda

Introduction to Trees
Binary Tree Def
Strict vs Complete
Array Representation
Properties

Non-linear Data Structures

Trees represent hierarchical relationships.
Unlike arrays/linked lists, trees have branching.
A tree consists of nodes and edges without cycles.

Binary Tree Definition

A tree where each node has at most 2 children.
Children are ordered: left and right.
Recursive def: empty or root + left subtree + right subtree.

Types of Binary Trees

Full/Strict Binary Tree: 0 or 2 children.
Perfect Binary Tree: All internal nodes have 2 children, all leaves at same level.
Complete Binary Tree: Filled left to right.

Complete Binary Tree

All levels are completely filled except possibly the last.
Last level is filled strictly left-to-right.
Why? Ensures minimal height and dense array packing.

Visualizing Complete Binary Tree

Level 0: 1 node
Level 1: 2 nodes
Level 2: up to 4 nodes (filled left to right)

Array Representation

Since there are no gaps, we use an array.
Root at index 0.
Left child of i: 2*i + 1
Right child of i: 2*i + 2
Parent of i: (i-1)//2

Code: Array Indexing

```
def left_child(i): return 2*i + 1
def right_child(i): return 2*i + 2
def parent(i): return (i-1)//2
```

This allows O(1) traversal to children/parent.

Properties & Height

For N nodes, a complete binary tree has height floor(log2(N)).
This guarantees O(log N) path lengths.
Essential for efficient operations in priority queues.

Summary

Binary trees: ¡= 2 children.
Complete BT: Dense, left-aligned.
Array representation is optimal for Complete BT.

Next Lecture

We will cover basic terminology:
Level number, degree, leaf nodes, etc.
Essential vocabulary for tree algorithms.

6.2 Basic Terms of Trees

Course: Data Structures
Unit 6: Binary Trees
Lecture 40

Agenda

Root and Edges
Level and Depth
Degree of a Node
Leaf and Internal Nodes
In-degree and Out-degree

Root and Edges

Root: The topmost node without a parent.
Edge: The connection between a parent and a child.
A tree with N nodes has exactly N-1 edges.

Level Number and Depth

Level: The root is at level 0. Its children are at level 1.
Depth of a node: Number of edges from root to the node.
Level and depth are often used interchangeably.

Calculating Depth

```
def get_depth(node):
    d = 0
    while node.parent:
        d += 1
        node = node.parent
    return d
```

Requires a parent pointer to trace upwards.

Height of a Node / Tree

Height of a node: Longest path from the node to a leaf.
Height of the tree: Height of the root node.
Leaves have height 0.

Degree of a Node

Degree: Total number of children a node has.
In a binary tree, the degree of any node is 0, 1, or 2.
Leaf node: Degree is 0.
Internal node: Degree ¿ 0.

In-degree and Out-degree

In-degree: Number of edges arriving at a node.
For a tree, root has in-degree 0; all others have in-degree 1.
Out-degree: Same as the degree of the node (number of children).

Code: Node Class with Degree

```
class Node:
    def __init__(self, val):
        self.val = val
        self.children = []
    def out_degree(self):
        return len(self.children)
```

For binary trees, children are explicitly ‘left‘ and ‘right‘.

Leaf and Internal Nodes

Leaf Node (External Node): A node with 0 out-degree.
Internal Node: A node with at least 1 child.
A strict binary tree with N leaves has N-1 internal nodes.

Summary

Level/Depth: Distance from root.
Height: Distance to furthest leaf.
Degree: Number of children.
Leaf: Node with no children.

Next Lecture

Binary Search Trees (BST).
How to order data within a binary tree for fast searching.
The left and right child properties.

6.3 Binary Search Tree (BST)

Course: Data Structures
Unit 6: Binary Trees
Lecture 41

Agenda

What is a BST?
The BST Property
Validating a BST
Finding Min and Max
Complexity Analysis

What is a Binary Search Tree?

A binary tree with an ordering property.
Allows for efficient searching, insertion, and deletion.
Combines the flexibility of linked lists with the fast access of arrays (binary search).

The BST Property

For any node N:

1. All keys in the left subtree are strictly LESS than N's key.
2. All keys in the right subtree are strictly GREATER than N's key.
3. Both left and right subtrees must also be BSTs.

Valid vs Invalid BST

Valid:
5
/ \{}
3 7
Invalid:
5
/ \{}
3 7
\{}
6 (Invalid if under 3, but 6 is ¿ 5, so it must be in right subtree of 5!)

Code: Node Definition

```
class BSTNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
```

Simple structure, powerful implications.

Finding the Minimum

Because of the BST property, smaller values are always to the left.
To find the minimum, simply follow ‘left‘ pointers until you reach a node with no left child.
Time complexity: O(h) where h is the tree height.

Code: Find Min

```
def find_min(root):
    current = root
    while current.left is not None:
        current = current.left
    return current.key
```

Iterative approach is extremely efficient.

Finding the Maximum

Similarly, larger values are always to the right.
To find the maximum, follow ‘right‘ pointers until you reach a node with no right child.
Time complexity: O(h).

Complexity Analysis

Average Case (balanced): Height h = O(log N). Operations take O(log N).
Worst Case (skewed): Height h = O(N). Operations take O(N).
A skewed BST degrades into a linked list.

Summary

BST imposes a strict left ¡ root ¡ right ordering.
Allows fast min/max lookups.
Performance depends heavily on tree balance.

Next Lecture

We will implement Insertion and Deletion in a BST.
Handling the three cases of deletion.
Re-wiring pointers correctly.

Insertion and Deletion in a BST

Course: Data Structures
Unit 6: Binary Trees
Lecture 42

Agenda

Insertion Algorithm
Insertion Code
Deletion: Case 1 (Leaf)
Deletion: Case 2 (One Child)
Deletion: Case 3 (Two Children)
Inorder Successor

Insertion Algorithm

Start at the root.
Compare the new key with the current node's key.
If new key ¡ current key, go left. If ¿ go right.
When you reach a NULL pointer, insert the new node there.

Code: Insertion

```
def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.val:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root
```

Recursive insertion is elegant.

Deletion Overview

Deletion is complex because it can break the BST structure.
We must handle three distinct cases based on the number of children the target node has:

1. Zero children (Leaf node)
2. One child
3. Two children

Deletion: Case 1 (Leaf Node)

The simplest case.
If the node to be deleted has no children, simply remove it.
Update the parent's pointer to NULL.

Deletion: Case 2 (One Child)

If the node has only one child, bypass the node.
Link the node's parent directly to the node's child.
The BST property is maintained since the subtree remains ordered.

Code: Deletion Cases 1 & 2

```
if root.left is None:
    return root.right
elif root.right is None:
    return root.left
```

Returning the non-null child effectively bypasses the current root.

Deletion: Case 3 (Two Children)

The most complex case. We cannot simply bypass the node.
Find the **Inorder Successor** (smallest node in the right subtree).
Replace the target node's value with the successor's value.
Recursively delete the successor from the right subtree.

Code: Deletion Case 3

```
# Node has two children
temp = find_min(root.right)
root.val = temp.val
root.right = delete(root.right, temp.val)
```

The successor will always have 0 or 1 child, reducing to Case 1 or 2.

Summary

Insertion always adds a leaf node in O(h) time.
Deletion has 3 cases.
Case 3 requires finding the inorder successor and recursive deletion.

Next Lecture

Searching for a node in a BST.
Iterative vs Recursive search.
Complexity analysis of search operations.

Searching a Node in a Binary Tree

Course: Data Structures
Unit 6: Binary Trees
Lecture 43

Agenda

Search in General Binary Tree
Search in BST
Recursive Search
Iterative Search
Performance Comparison

Search in General Binary Tree

Without the BST property, we must traverse the entire tree.
Use BFS or DFS.
Time complexity: O(N) because we might check every node.
Space complexity: O(N) for recursion stack or queue.

Search in a BST

The BST property acts like Binary Search in an array.
Compare target with current node.
If target == node.val, return node.
If target ¡ node.val, search left subtree.
If target ¿ node.val, search right subtree.

Code: Recursive BST Search

```
def search_rec(root, key):
    if root is None or root.val == key:
        return root
    if key < root.val:
        return search_rec(root.left, key)
    return search_rec(root.right, key)
```

Clean, but uses O(h) stack space.

Iterative Search in BST

We can avoid the recursion stack space overhead.
Use a simple while loop.
Update the current pointer based on comparisons.
Space complexity: O(1).

Code: Iterative BST Search

```
def search_iter(root, key):
    curr = root
    while curr is not None and curr.val != key:
        if key < curr.val:
            curr = curr.left
        else:
            curr = curr.right
    return curr
```

Simple and highly efficient.

Complexity Analysis

Time Complexity:
- Best Case: O(1) (target is root)
- Average Case: O(log N) (balanced tree)
- Worst Case: O(N) (skewed tree)

Space Complexity: O(h) recursive, O(1) iterative.

Handling Duplicates

What if the BST allows duplicate values?
Convention: Store duplicates strictly in the left (or right) subtree.
Search might need to continue until a leaf to find *all* duplicates.
Alternative: Store a frequency counter in the node.

Node with Frequency

```
class Node:
    def __init__(self, key):
        self.key = key
        self.freq = 1
        self.left = None
        self.right = None
```

Saves space and simplifies search logic.

Summary

BST search is analogous to binary search.
Iterative approach uses O(1) auxiliary space.
Worst-case time is O(N) if the tree is unbalanced.

Next Lecture

Tree Traversal techniques.
Visiting every node in a specific order.
Preorder, Inorder, and Postorder traversals.

6.4 Tree Traversal

Course: Data Structures
Unit 6: Binary Trees
Lecture 44

Agenda

What is Traversal?
BFS vs DFS
Level Order Traversal (BFS)
Implementation of BFS
Introduction to DFS

What is Tree Traversal?

Process of visiting each node in a tree data structure, exactly once.
Used to print, update, or serialize the tree.
Unlike arrays (linear), trees can be traversed in multiple ways.

Breadth-First vs Depth-First

Breadth-First Search (BFS): Explores level by level, top to bottom, left to right.
Depth-First Search (DFS): Explores as far down a branch as possible before backtracking.
Both visit all N nodes, taking O(N) time.

Level Order Traversal (BFS)

Visits nodes level by level.
Root is visited first (Level 0).
Then all children of root (Level 1).
Requires a Queue (FIFO) data structure.

Algorithm: Level Order

1. Create an empty queue Q.
2. Enqueue the root node.
3. Loop while Q is not empty:
a. Dequeue a node and visit it.
b. Enqueue its left child (if exists).
c. Enqueue its right child (if exists).

Code: Level Order

```
from collections import deque
def level_order(root):
    if not root: return
    q = deque([root])
    while q:
        node = q.popleft()
        print(node.val, end=' ')
        if node.left: q.append(node.left)
        if node.right: q.append(node.right)
```

Using deque for O(1) pops.

Complexity of Level Order

Time Complexity: O(N) since each node is enqueued and dequeued exactly once.
Space Complexity: O(W) where W is the maximum width of the tree.
For a complete binary tree, $W = N/2$, so space is O(N).

Depth-First Search (DFS)

Explores down to the leaves before exploring siblings.
Uses a Stack (LIFO) data structure.
Can be implemented iteratively (explicit stack) or recursively (call stack).
Three main variants: Preorder, Inorder, Postorder.

Why Different Traversals?

Level order is good for finding shortest paths or printing tree visually.
DFS traversals are good for evaluating expressions, flattening trees, or deleting trees.
The choice depends on the specific application requirements.

Summary

Traversal visits every node.
BFS (Level Order) uses a Queue.
DFS uses a Stack (or recursion).
Time is O(N) for both.

Next Lecture

Deep dive into DFS variants.
Inorder, Preorder, Postorder.
Applications of Binary Trees.

DFS Traversals and Applications

Course: Data Structures
Unit 6: Binary Trees
Lecture 45

Agenda

Inorder Traversal
Preorder Traversal
Postorder Traversal
Expression Trees
Applications of Binary Trees

Inorder Traversal (Left, Root, Right)

Algorithm:
1. Traverse the left subtree in Inorder.
2. Visit the root.
3. Traverse the right subtree in Inorder.

Crucial Property: Inorder traversal of a BST yields elements in sorted (ascending) order.

Code: Inorder

```
def inorder(root):
    if root:
        inorder(root.left)
        print(root.val, end=' ')
        inorder(root.right)
```

Simple and elegant recursion.

Preorder Traversal (Root, Left, Right)

Algorithm:
1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

Used for creating a copy of the tree, or prefix expressions.

Code: Preorder

```
def preorder(root):
    if root:
        print(root.val, end=' ')
        preorder(root.left)
        preorder(root.right)
```

Root is processed first.

Postorder Traversal (Left, Right, Root)

Algorithm:
1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

Used for deleting a tree (must delete children before parent) or postfix expressions.

Code: Postorder

```
def postorder(root):
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.val, end=' ')
```

Safest way to delete a tree.

Expression Trees

Binary trees can represent arithmetic expressions.
Leaf nodes are operands (e.g., 5, x).
Internal nodes are operators (e.g., +, *).
Inorder traversal gives infix expression (needs parentheses).
Postorder gives postfix (RPN).

Applications of Binary Trees

Binary Search Trees: Fast searching, sets, maps.
Heaps (Complete BT): Priority Queues, Heap Sort.
Syntax Trees: Used by compilers to parse expressions.
Huffman Coding Tree: Data compression algorithms.
Tries / Routing Tables: Storing prefix data.

Summary

Inorder (BST) -¿ Sorted Output.
Preorder -¿ Tree Copy.
Postorder -¿ Tree Deletion.
Binary trees form the backbone of many advanced data structures.

End of Unit 6

This concludes Unit 6 on Binary Trees.
Next unit will cover advanced trees like AVL or Graphs.
Review these concepts for the exam.